# In Built STL

* Summation:

accumulate(first, last, init, op);  
example:

int sum = accumulate(v.begin(), v.end(), 0);

* Permutation:

std::next\_permutation(v.begin(), v.end());

Returns true if the container could be rearranged to the to the lexicographical larger permutation.

Returns false otherwise.

std::prev\_permutation(v.begin(), v.end());

Returns true if the container could be rearranged to the to the lexicographical smaller permutation.

Returns false otherwise.

<https://www.geeksforgeeks.org/stdnext_permutation-prev_permutation-c/>

**Links:**

1. [For short Overview](https://youtu.be/Cg7SI0WtmXY?si=aPtORzMfoh7sWBqn)
2. [For better Knowledge](https://youtu.be/6zhGS79oQ4k?si=61r9scRW1gKfQhBD)

**Notes:**

1. Lower bound: finds smallest index such that arr[indx]>=n and if the last element of the array smaller than n then this function will provide an address next index of last element( arr.end() or &arr[n+1])  
   Vector: in = lower\_bound(v.begin(), v.end(), x) – v.begin()  
   array: int \*in = lower\_bound(arr, arr+n, x)  
   set: auto it = s.lower\_bound(x)  
   map: auto it = mp.lower\_bound (key)  
   Pair: indx = lower\_bound(p.begin(), p.end(), find\_p) – p.begin()  
   in Vector of Pairs **lower\_bound()** for **pair(x, y)** will return an iterator pointing to the position of pair whose:  
   #first value is greater than **x**#first value is equal to  **x** and second value is greater than equals to **y**.
2. Upper bound: finds smallest index such that arr[indx]>n  
   Vector: in = upper\_bound(v.begin(), v.end(), x) – v.begin()  
   array: int \*in = upper\_bound(arr, arr+n, x)  
   set: auto it = s.upper\_bound(x)  
   map: auto it = mp.upper\_bound(key)  
   Pair: indx = upper\_bound(p.begin(), p.end(), find\_p) – p.begin()  
   in Vector of Pairs **upper\_bound()** for **pair(x, y)** will return an iterator pointing to the position of the pair whose

#First value is equal to x and the second value is greater than y, or

#Whose first value is greater than x.

# **Set**

[**Ordered set:**](https://www.programiz.com/cpp-programming/set) where we can store unique values in ordered manner.

* Duplicates are not allow
* We are not able to modify a value but add and remove operation is allow
* By default stores in increasing order but we can change it
* We are not able to access like by using index instead we need use iterator

**Need to include:** #include<set>

**Structure:** set<datatype> name; set<datatype> name={val, val}; or

set<dtype, greater<dtype>> name;

**Functions:**

* s.insert(x) – for inserting value x into the set
* s.insert({x, y, z}) – for inserting multiple value
* s.erase(x) – for erasing value x from the set
* s.clear() - for clearing full set
* s.empty() – for checking is set empty or not if yes then it returns 1 or true
* s.size() – for finding size of the set
* s.count(x) – returns 1 if exist else 0
* s.find(x) – it returns an iterator(memory address) where x exist if it does not exist in the set then it retuns s.end().

**Key features**:

* Stores unique elements.
* Elements are automatically sorted in ascending order by default.
* Time complexity for search, insertion, and deletion: O(log n) (based on binary search trees, usually red-black trees).

[**Multi set:**](https://www.programiz.com/cpp-programming/multiset) everything is same as set but we can store here duplicate value

Structure: multiset<dtype> name

**Key features**:

* Stores sorted elements, but duplicates are allowed.
* Time complexity for search, insertion, and deletion: O(log n) (similar to set, based on balanced binary trees).

[**Unordered set:**](https://www.programiz.com/cpp-programming/unordered-set) Here data stores without following any order everything Is same

**Need to include:** #include<unordered\_set>

Structure: unordered\_set<dtype> name;

**Key features**:

* Stores unique elements in no particular order.
* Average time complexity for search, insertion, and deletion: O(1) (using hash tables), but worst case is O(n).

# Stack

Need to include: #include<stack>

Structure: stack<datatype> stackname;

**Functions of stack:**

* push(): for pushing element in the top of stack  
  st.push(“hola”);
* pop(): for removing top element from the stack  
  st.pop();
* top(): for getting top element of our stack  
  text = st.top();
* size(): for knowing size of the stack  
  len = st.size();
* empty(): to know is our stack empty or not   
  if(!st.empty) cout << “NO”;

# Priority Queue

It is kind of stack but here elements stores in sorted manner.

#include<queue>

priority\_queue<type> pq; for storing data in decreasing order

priority\_queue<type, vector<type>, greater<int>> pq; for storing data in increasing order

**Methods:**

* push()
* pop()
* top()
* size()
* empty()

# Deque

Deque is special type of Queue where we can access from both front and back.

#include<deque>

deque<type> dq

**Methods:**

* push\_back()
* push\_front()
* pop\_back()
* pop\_front()
* front()
* back()
* at()
* size()
* empty()

# Min && Max

\*min\_element(v.begin(), v.end())

\*max\_element(v.begin(), v.end())

**For finding both using one call:**

auto minmax = minmax\_element(v.begin(), v.end()); [it returns a pair]

cout << \*minmax.first;

cout << \*minmax.second;

**Complexity:**

Time: O(n)

Space: O(1)